MODULE 4

WORKING WITH FUNCTIONS

CO - Students will be able to apply modular programming approach using functions

FUNCTION

- A function is a named part of a program that can be invoked from other part of the program.
- That is, a function is a self-contained block of statement that performs a coherent task of some kind.
- Every C program can be thought of as a collection of these functions.
- A function can be classified in to two categories.

❖**Library functions:** These are pre-defined. Library functions are not required to be written by us.

- Examples are **printf () ,scanf (), sqrt (), cos (), strcat ()** etc.
- This library of functions is present on the disk and is written for us by people who write compilers for us.
- Almost always a compiler comes with a library of standard functions. The procedure of calling both types of functions is exactly same.

❖**User defined functions:** This type of functions has to be developed by the user at the time of writing a program.

- **main ()** is an example of user defined function.

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 3

➢A function has three parts

1. **Function declaration** **Synax: type function_name (argument list);**
2. **Function call** **Syntax: function_name(argument list);**
3. **Function definition**

❖**Function definition**

Syntax:

```
type function-name (parameter/argument list)
    {
       Local variable declarations;
       executable statement 1;
       executable statement 2;
          ……….
       return statement;
    }
```

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 4

- **type function-name (parameter list)** is called as **function header**
- The statement with in the opening and closing braces are function body.
- Semicolon is not used at the end of function header.
- A function must be declared before the main () function.
- Function calling is done with in the main () function.
- Parameter is also called as arguments.

A SIMPLE FUNCTION

```
#include<stdio.h>
#include<conio.h>
void message(); /* function declaration*/
void main( )
{
    message( ) ;  /* function call*/
    printf ( "Hai...It's Monday!" ) ;
    getch();
}

void message( )  /* function definition*/
{
    printf ( "Hai…It's Tuesday" ) ;
}
```

OUTPUT

Hai…It's Tuesday

Hai…It's Monday!

- Here, **main()** itself is a function and through it we are calling the function **message()**.
- **main()** becomes the 'calling' function, whereas **message()** becomes the 'called' function.
- Any C program contains at least one function
- If a program contains only one function, it must be **main ()**.
- If a C program contains more than one function, then one (and only one) of these functions must be **main ()**, because program execution always begins with **main ()**.
- There is no limit on the number of functions that might be present in a C program
- Each function in a program is called in the sequence specified by the function calls in main().

- After each function has done its thing, control returns to **main ( )**.

  When **main ( )** runs out of function calls, the program ends.
- C program is a collection of one or more functions
- A function gets called when the function name is followed by a semicolon. For example,
- A function can call itself. Such a process is called **'recursion'.**
- A function can be called from other function, but a function cannot be defined in another function

Small Program Using Return Statement

```
#include<stdio.h>
#include<conio.h>
int mul(int, int);      /* function declaration*/
void main ()
{
      int y,a,b;
     printf("Enter two integer values");
     scanf("%d%d",&a,&b);
      y=mul (a, b);     /* function call using actual parameters a and b*/
      printf ("%d\n",y);
      getch();
}

int mul(int x, int y)  /* function definition using formal parameters(dummy variables) */
{
    int p; /* local variable declaration*/
    p=x*y;
    return(p);
}
```


- The **return** statement serves two purposes:
- On executing the **return** statement it immediately transfers the control back to the calling program.
- It returns the value present in the parentheses after **return**, to the calling program. In the above program the value of sum of three numbers is being returned
- All the following are valid **return** statements.

```
return ( a ) ;
return ( 23 ) ;
return ( 16.94 ) ;
return ;
```

- If we want that a called function should not return any value, in that case, we must mention so by using the keyword **void** as shown below.

```
void display( )
{
   printf ( "\nHello every one" ) ;
}
```

- In the absence of return statement, the closing brace act as a void return.


```
int a,b;        - Variable declaration

int a[20];      - Array declaration (One Dimensional)
int a[4][4];    - Array declaration (Two Dimensional)
char a[25];     - String declaration
```

Function Declarations

```
void  add(int,int);
int add (int,float);
float add(float,float);
void add ();
```

❖Example for Function without argument and return value

```
#include<stdio.h>
#include<conio.h>
void sum();
void main()
{
   printf("\nGoing to calculate the sum of two numbers:");
   sum();
   getch();
}
void sum()
{
   int a,b;
   printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   printf("The sum is %d",a+b);
}
```

OUTPUT

Going to calculate the sum of two numbers:

Enter two numbers

10 24

The sum is 34

❖Example for Function without argument and with return value

```
#include<stdio.h>
#include<conio.h>
int square();
void main()
{
   float area
   printf("Going to calculate the area of the square\n");
   area = square();
   printf("The area of the square: %f\n",area);
   getch();
}
int square()
{
   float side;
   printf("Enter the length of the side in meters: ");
   scanf("%f",&side);
   return side * side;
}
```

OUTPUT

Going to calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

❖Example for Function with argument and without return value

```
#include<stdio.h>
#include<conio.h>
void average(int, int, int, int, int);
void main()
{
   int a,b,c,d,e;
   printf("\nGoing to calculate the average of five numbers:");
   printf("\nEnter five numbers:");
   scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
   average(a,b,c,d,e);
   getch();
}
void average(int a, int b, int c, int d, int e)
{
   float avg;
   avg = (a+b+c+d+e)/5;
   printf("The average of given five numbers : %f",avg);
}
```

OUTPUT

Going to calculate the average of five numbers:

Enter five numbers: 10 20 30 40 50

The average of given five numbers: 30.000000

❖Example for Function with argument and with return value

```
#include<stdio.h>
#include<conio.h>
int sum(int, int);
void main()
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
   getch();
}
int sum(int a, int b)
{
   return a+b;
}
```

OUTPUT

Going to calculate the sum of two numbers:

Enter two numbers: 10 20

The sum is : 30

PASS BY VALUE (CALL BY VALUE)

- In this method the 'value' of each of the actual arguments (arguments in the function call statement) in the calling function is copied into corresponding formal arguments (arguments in the function definition section) of the called function.
- The function creates its own copy of argument values and then uses them.
- With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.
- That is, any change in the formal parameter is not reflected back to actual parameters.

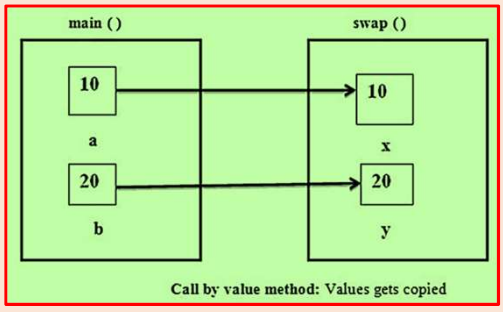

```
#include<stdio.h>
#include<conio.h>
void swap(int, int);  /* function declaration*/
void main( )
{
    int a = 10, b = 20 ;
    swap ( a, b ) ;  /* function call*/
    printf ( "\na = %d b = %d", a, b ) ;
    getch();
}

void swap ( int x, int y ) /* function definition*/
{
    int temp ;
    temp = x ;
     x = y;
    y = temp;
    printf ( "\nx = %d y = %d", x, y ) ;
}
```

OUTPUT

X=20 y=10

a=10 b=20

❖ Note that values of **a** and **b** remain unchanged even after exchanging the values of **x** and **y**.



RECURSION

- In C, it is possible for the functions to call themselves. A function is called 'recursive' if a statement within the body of a function calls the same function.
- A function can call itself. Such a process is called '**recursion**'.

❖**EXAMPLE**

- Factorial of a number is the product of all the integers between 1 and that number. For example, 4 factorial is 4 * 3 * 2 * 1. This can also be expressed as 4! = 4 * 3! Where '!' stands for factorial.
- Thus factorial of a number can be expressed in the form of itself. Hence this can be programmed using recursion.


```c
#include <stdio.h>
#include <conio.h>
int fact (int);
void main()
{
   int n,f;
   printf("Enter the number whose factorial you want to calculate?");
   scanf("%d",&n);
   f = fact(n);
   printf("factorial = %d",f);
   getch();
}
int fact(int n)
{
   if (n==0)
     {
      return 0;
     }
   else if ( n == 1)
    {
     return 1;
    }
   else
    {
     return n*fact(n-1);
    }
 }
```

OUTPUT

Enter the number whose factorial you want to calculate?

5

```
return 5 * factorial(4) = 120
    return 4 * factorial(3) = 24
        return 3 * factorial(2) = 6
            return 2 * factorial(1) = 2
                return 1 * factorial(0) = 1
```

1*2*3*4*5 = 120

PASSING 1D ARRAY ELEMENTS TO A FUNCTION

```
#include <stdio.h>
#include <conio.h>
void display(int);
void main( )
{
  int i ;
  int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
  for ( i = 0 ; i <= 6 ; i++ )
    display ( marks[i] ) ;
  getch();
}

void display ( int m )
{
   printf ( "%d ", m ) ;
}
```

OUTPUT

55 65 75 56 78 78 90

since at a time only one element is being passed, this element is collected in an ordinary integer variable m, in the function display( ).

PASSING 1D ARRAY TO A FUNCTION (Second Method)

```
#include<stdio.h>
#include<conio.h>
float largest(float b[ ],int n);      // function declaration
void  main ( )
{
 float a[5],i ;
float p;
for (i=0;i<5;i++)
scanf ("%f",&a[i]);
p=largest (a, 5);       // function call
printf("The largest value is%f"p);
getch();
}
```

```
float largest (float b[ ], int n)    //function definition
{
int i;
float max;
max=b[0];
for(i=1;i<n;i++)
  if(max<b[i])
    max=b[i];
return(max);
}
```

PASSING 2D ARRAY TO A FUNCTION

```
#include <stdio.h>
#include <conio.h>
void show(int q[ ][4], int row, int col);
void main( )
{
   int a[3][4] =  {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 6} ;
   show ( a, 3, 4 ) ;
   getch();
}

show ( int q[ ][4], int row, int col )
{
   int i, j ;
   for ( i = 0 ; i < row ; i++ )
     {
        for ( j = 0 ; j < col ; j++ )
          printf ( "%d ", q[i][j] ) ;
          printf ( "\n" ) ;
     }
    printf ( "\n" ) ;
 }
```

OUTPUT

```
1 2 3 4
5 6 7 8
9 0 1 6
```

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 23

STORAGE CLASSES

- Storage classes provide the information about variable's location and visibility. Variable's storage class tells us
  - Where the variable would be stored.
  - What will be the initial value of the variable, if initial value is not specifically assigned? (i.e. the default initial value).
  - What is the scope of the variable; i.e. in which functions the value of the variable would be available.
  - What is the life of the variable; i.e. how long would the variable exist.

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 24

- **Scope of variable** determines over what region of the program a variable is actually available for use.
- **Visibility** refers to the accessibility of a variable from the memory
- **Longevity** refers to the period during which a variable retains a given value during execution of a program.
- **Life time** of a variable is the duration of time in which a variable exists in the memory during execution.

❖**There are four storage classes in C:**

➢Automatic storage class
➢Register storage class
➢Static storage class
➢External storage class

❖Automatic Storage Class

- The features of a variable defined to have an automatic storage class are as under:

Storage	**– Memory.**
Default initial value	**– An unpredictable value, which is often called a garbage value.**
Scope	**– Local to the block in which the variable is defined.**
Life	**– Till the control remains within the block in which the variable is defined.**

- A variable declared inside a function by default, automatic. They are created when the function is called and destroyed automatically when the function is exited.
- We can write explicitly as **auto int number;**

❖Register Storage Class

- The features of a variable defined to be of register storage class are as under:

Storage	-	CPU registers.
Default initial value	-	Garbage value.
Scope	-	Local to the block in which the variable is defined.
Life	-	Till the control remains within the block in which the variable is defined.

- We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory. Register access is faster than memory access.

```
register int count;
```


❖Static Storage Class

- The features of a variable defined to have a static storage class are as under:

Storage	–	Memory
Default initial value	–	Zero
Scope	–	Local to the block in which the variable is defined.
Life	–	Value of the variable persists between different function calls.

- Static variable initialized once when the program is compiled. The value of static variable cannot change.

```
static int x;
```

❖External Storage Class

- The features of a variable whose storage class has been defined as external are as follows:

Storage	**– Memory.**
Default initial value	**– Zero.**
Scope	**– Global.**
Life	**– As long as the program's execution doesn't come to an end.**

- External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them

extern int y;

AUTO & STATIC – A COMPARISON

```
main( )                          main( )
{                                {
    increment( ) ;                   increment( ) ;
    increment( ) ;                   increment( ) ;
    increment( ) ;                   increment( ) ;
}                                }

increment( )                     increment( )
{                                {
    auto int i = 1 ;                 static int i = 1 ;
    printf ( "%d\n", i ) ;           printf ( "%d\n", i ) ;
    i = i + 1 ;                      i = i + 1 ;
}                                }
```

The output of the above programs would be:

1	1
1	2
1	3

STRUCTURE

- A structure is a collection of variables of different data types that are referenced by a common name.
- It is a mechanism for packing data of different types.
- A structure contains a number of data types grouped together.
- These data types may or may not be of the same type.

structure declaration Syntax

```
struct <structure name>
{
  structure element 1 ;
  structure element 2 ;
  structure element 3 ;
  ......
  ......
} ;
```

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 31

- Once the new structure data type has been defined one or more variables can be declared to be of that type.

For example,

```
struct book
{
   char name ;
   float price ;
   int pages ;
} ;
struct book b1, b2, b3 ;
```

is same as

```
struct book
{
   char name ;
   float price ;
   int pages ;
} b1, b2, b3 ;
```

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 32

❖Structure Initialization

- Like primary variables and arrays, structure variables can also be initialized where they are declared.

```
struct book
{
    char name[10] ;
    float price ;
    int pages ;
} ;
struct book b1 = { "Chemistry", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
```

- The closing brace in the structure type declaration must be followed by a semicolon.

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 33

ACCESSING STRUCTURE ELEMENTS

- In arrays we can access individual elements of an array using a subscript. Structures use a different scheme.
- They use a **dot (.) operator**. So to refer to pages of the structure defined in our sample program we have to use,

b1.pages

Similarly, to refer to price we would use,

b1.price

```
struct book
{
   char name ;
   float price ;
   int pages ;
} ;
struct book b1, b2, b3 ;
```

- Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 34

```
struct book-bank
{
char title[20];
char author[15];
int pages ;
float price;
} book1, book2, book3;
```

We can assign values to the members of book1.

```
strcpy (book1.title,"C++");
strcpy (book1.author,"XYZA");
book1.pages=290;
book1.price=320.50;
```

We can also use scanf to give the values through the keyword.

```
scanf ("%s\n",book1.title);
scanf ("%d\n",&book1.pages);
```


ARRAY OF STRUCTURES

- **Array of structures:** an array of similar data types which themselves are a collection of dissimilar data types.

```
struct student
{
  int x;
  float m;
} a [3] = {{10,3.5}, {5, 1.2}, {25, 0.07}};
```

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct book
    {
      char name ;
      float price ;
      int pages ;
    } ;
  struct book b[200] ;
  int i ;
  for ( i = 0 ; i <= 199 ; i++ )
   {
    printf ( "\nEnter name, price and pages " ) ;
    scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
   }
  for ( i = 0 ; i <= 199 ; i++ )
    printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;
  getch();
}
```


STRUCTURE ASSIGNMENT

- The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator.
- It is not necessary to copy the structure elements piece-meal.
- Obviously, programmers prefer assignment to piece-meal copying.
- This is shown in the following example.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct employee
     {
       char name[20] ;
       int age ; float salary ;
     } ;
   struct employee e1 = { "Sanjay", 30, 5500.50 } ;
   struct employee e2, e3 ;
 /* piece -meal copying */
   strcpy ( e2.name, e1.name ) ;
   e2.age = e1.age;
   e2.salary = e1.salary;
/* copying all elements at one go */
   e3 = e2;
   printf ( "\n%s %d %f", e1.name, e1.age, e1.salary ) ;
   printf ( "\n%s %d %f", e2.name, e2.age, e2.salary ) ;
   printf ( "\n%s %d %f", e3.name, e3.age, e3.salary ) ;
   getch();
}
```

OUTPUT

Sanjay 30 5500.500000
Sanjay 30 5500.500000
Sanjay 30 5500.500000

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 39

PASSING STRUCTURE TO A FUNCTION

```
#include<stdio.h>
#include<conio.h>
 struct book
   {
     char name[25] ;
     char author[25] ;
     int callno ;
   } ;
void main( )
  {
    struct book b1 = { "Programming in C ", "James", 9952489217 } ;
    display ( b1 ) ;
    getch();
  }
display ( struct book b )
{
     printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```

OUTPUT

Programming in C James 9952489217

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 40

UNION

- The syntax is same as structures. In structure each member has its own storage location, whereas all the members of a union use the same location.
- It can handle only one member at a time.
- Union may be used in all places where a structure is allowed.
- To access a union member, we can use the same syntax that we use for structure members. That is,

  code.m
  code.x
  code.c all are valid.

```
union item
  {
    int m;
    float x;
    char c;
  } code;
```


- The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.
- In the above declaration, the member x requires 4 byte memory. So only the 4 byte memory is allocated.